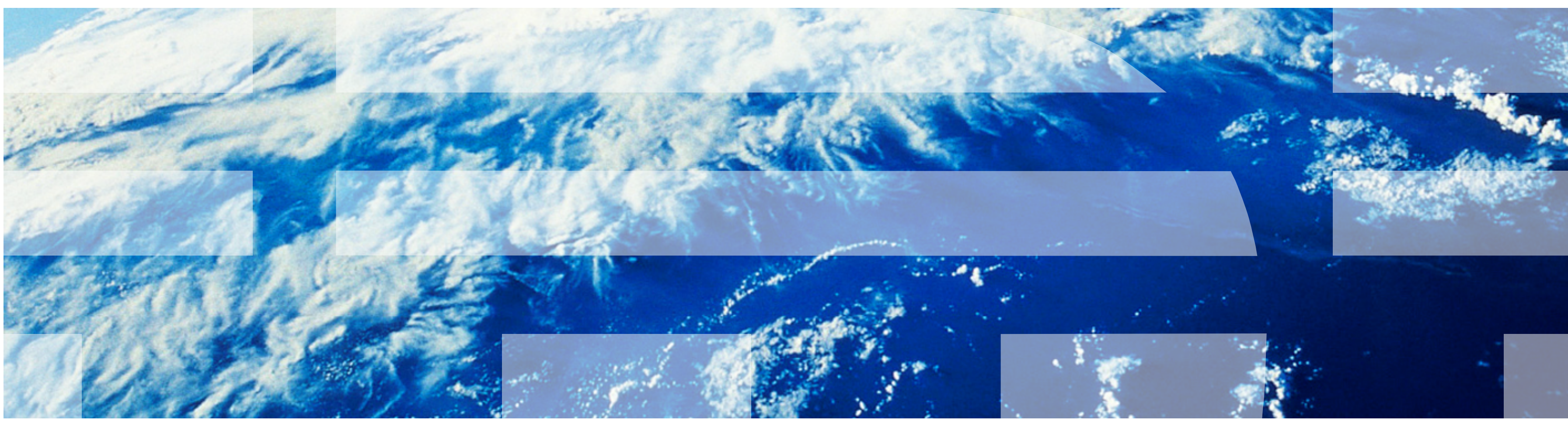
Adapted from Database System Concepts, 7th edition, Silberschatz, Korth and Sudarshan

Computer Systems for Data Science
Topic 5

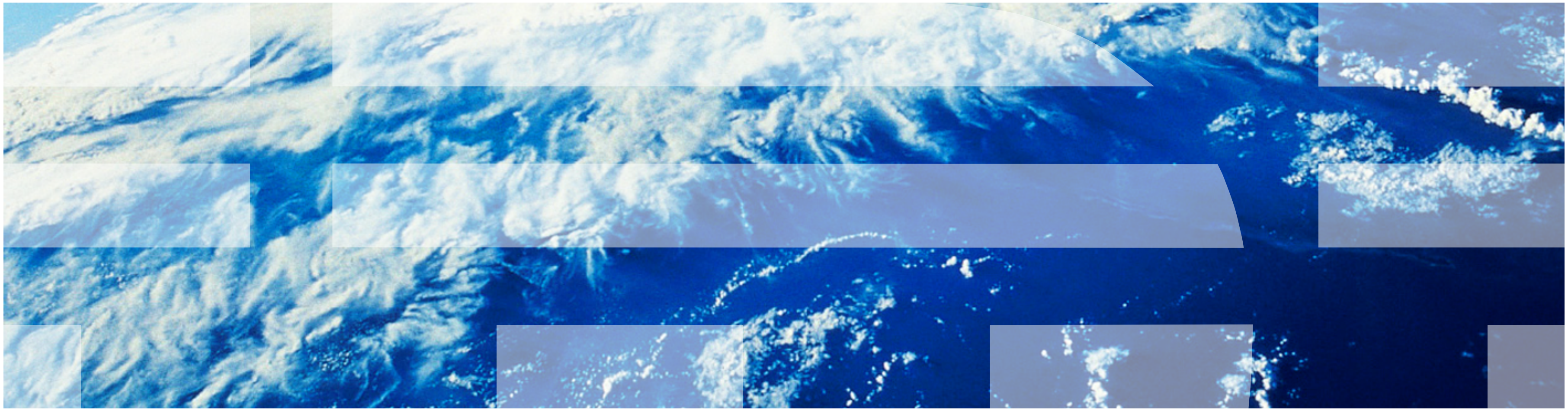
Distributed File Systems

Distributed Databases

Two Phase Commit



Partitioning and Replication



Introduction

- Data storage needs are growing increasingly large
  - user data at web-scale
    - 100’s of millions of users, petabytes of data
  - transaction data are collected and stored for analysis.
  - multimedia objects like images/videos
- Parallel storage system requirements
  - storing large volumes of data
  - processing time-consuming decision-support queries
  - providing high throughput for transaction processing
  - Very high demands on **scalability** and **availability**

Parallel/Distributed Data Storage History

- 1980/1990s
  - Distributed database systems with tens of nodes
- 2000s:
  - Distributed file systems with 1000s of nodes
    - Millions of Large objects (100’s of megabytes)
    - Web logs, images, videos, …
    - Typically create/append only
  - Distributed data storage systems with 1000s of nodes
    - Billions to trillions of smaller (kilobyte to megabyte) objects
    - Social media posts, email, online purchases, …
    - Inserts, updates, deletes
  - **Key-value stores**
- 2010s: Distributed database systems with 1000s of nodes

Storage Parallelism

- Reduce the time required to retrieve data from disk by partitioning the relations on *multiple disks*, on *multiple* ***nodes*** (computers)
  - Our description focuses on parallelism across nodes
  - Same techniques can be used across disks on a node
- **Horizontal partitioning** – tuples of a relation are divided among many nodes such that some subset of relation resides on each node.
  - Contrast with **vertical partitioning**, e.g. *r*(*A,B,C,D*) with primary key *A* into r1(*A,B*) and r2(*A,C,D*)

Storage Parallelism

- Partitioning techniques (number of nodes = n):

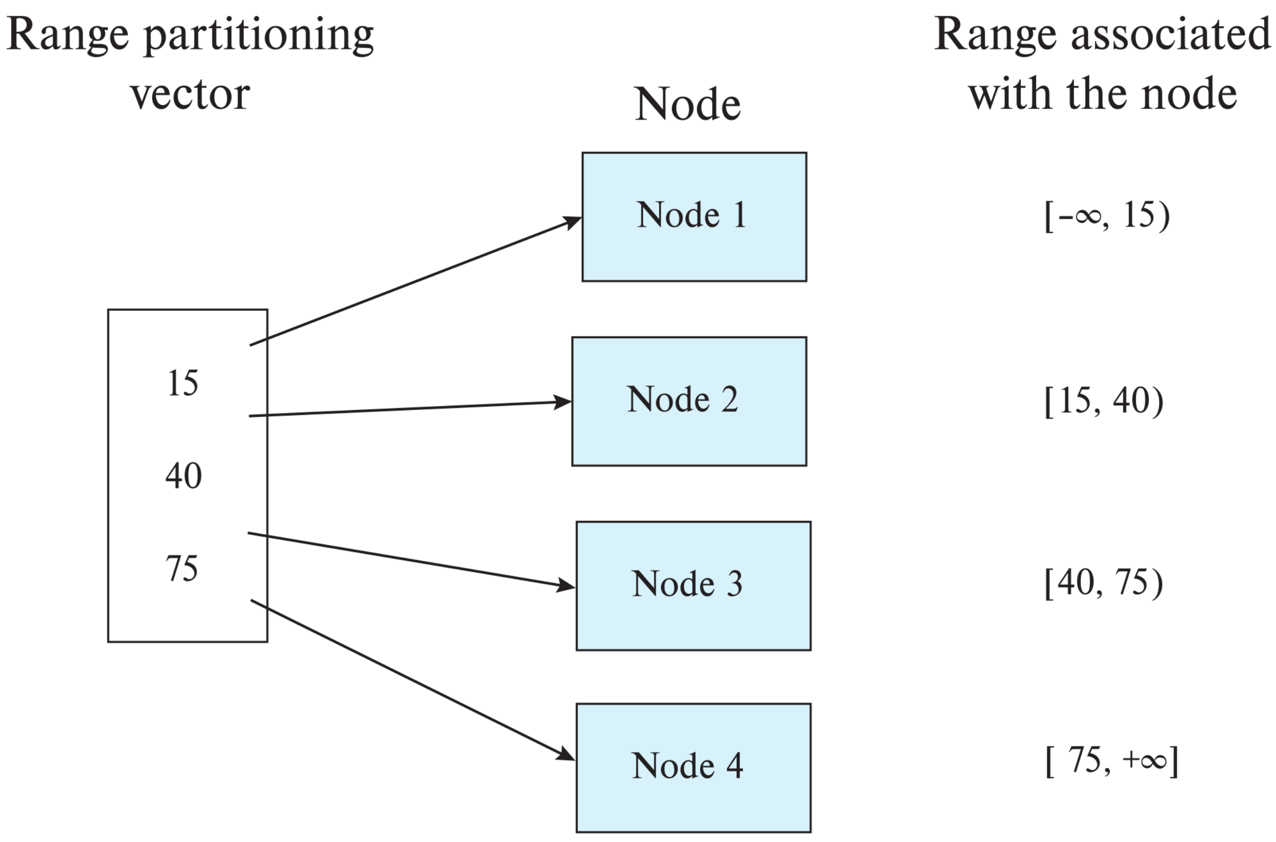
  Round-robin:

  Send the i^{th} tuple inserted in the relation to node i mod n.

  Hash partitioning:

  - Choose one or more attributes as the partitioning attributes.
  - Choose hash function h with range $0 \ldots n - 1$
  - Let i denote result of hash function h applied to the partitioning attribute value of a tuple. Send tuple to node i.

Range Partitioning



Storage Parallelism (Cont.)

Partitioning techniques (cont.):

- **Range partitioning**:
  - Choose an attribute as the partitioning attribute.
  - A partitioning vector [v_o, v_1, ..., v_{n-2}] is chosen.
  - Let v be the partitioning attribute value of a tuple. Tuples such that $v_i \leq v_{i+1}$ go to node $I + 1$. Tuples with $v < v_0$ go to node 0 and tuples with $v \geq v_{n-2}$ go to node n-1.

  E.g., with a partitioning vector [5,11]
    - a tuple with partitioning attribute value of 2 will go to node 0,
    - a tuple with value 8 will go to node 1, while
    - a tuple with value 20 will go to node2.

Comparison of Partitioning Techniques

- Evaluate how well partitioning techniques support the following types of data access:
  1. Scanning the entire relation.
  2. Locating a tuple associatively – **point queries**.
     - E.g., $r.A = 25$.
  3. Locating all tuples such that the value of a given attribute lies within a specified range – **range queries**.
     - E.g., $10 \leq r.A < 25$.
- Do above evaluation for each of
  - Round robin
  - Hash partitioning
  - Range partitioning

Comparison of Partitioning Techniques

Round robin:

- Best suited for sequential scan of entire relation on each query.
  - All nodes have almost an equal number of tuples; retrieval work is thus well balanced between nodes.
- Disadvantage: all queries involving multiple tuples must be processed at all nodes

Hash partitioning:

- Similar to round robin: good for sequential access
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between nodes
  - Less evenly distributed than round robin (uniform random is not the same as round robin)
- Disadvantage: all queries involving multiple tuples must be processed at all nodes

Comparison of Partitioning Techniques

Range partitioning:

- Provides data clustering by partitioning attribute value.
  - Good for sequential access
  - Good for point queries on partitioning attribute: only one node needs to be accessed.
- For range queries on partitioning attribute, one to a few nodes may need to be accessed
  - Remaining nodes are available for other queries.
  - Good if result tuples are from one to a few blocks.
  - But if many blocks are to be fetched, they are still fetched from one to a few nodes, and potential parallelism in disk access is wasted
    - Example of **execution skew**.

Handling Small Relations

- Partitioning not useful for small relations which fit into a single disk block or a small number of disk blocks
  - Instead, assign the relation to a single node, or
  - Replicate relation at all nodes
- For medium sized relations, choose how many nodes to partition across based on size of relation
- Large relations typically partitioned across all available nodes.

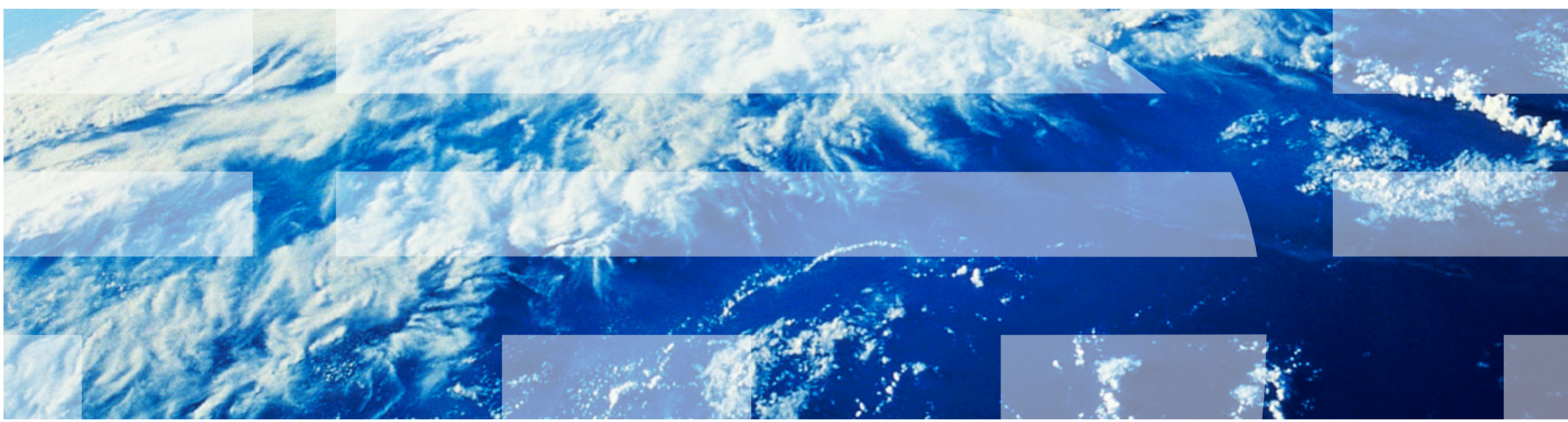
Types of Skew

- **Data-distribution skew:** some nodes have many tuples, while others may have fewer tuples. Could occur due to
  - **Attribute-value skew**.
    - Some partitioning-attribute values appear in many tuples
    - All the tuples with the same value for the partitioning attribute end up in the same partition.
    - Can occur with range-partitioning and hash-partitioning.
  - **Partition skew**.
    - Imbalance, even without attribute –value skew
    - Badly chosen range-partition vector may assign too many tuples to some partitions and too few to others.
    - Less likely with hash-partitioning

Types of Skew (Cont.)

- Note that **execution skew** can occur even without data distribution skew
  - E.g. relation range-partitioned on date, and most queries access tuples with recent dates
- Data-distribution skew can be avoided with range-partitioning by creating **balanced range-partitioning vectors**

Lecture 8



Recap of lecture 7

- **Key-value store and database example (RocksDB + MyRocks)**
  - Key-value stores
    - Much more limited interface than SQL (NoSQL)
    - Usually: get, put, update, delete, sometimes get_range, multiget
    - Often SQL/ACID databases are built on top of key-value stores
  - RocksDB
    - Writes always go to memory: buffer writes to flash
    - Data on disk is stored in immutable files
    - Log-structure merge tree data structure compacts files, removes stale data (old/deleted key-value pairs)
    - Uses bloom filters
    - Multi-level index: outer index for each level (sparse index, in levels 1-N have distinct and sorted key-ranges), inner and dense index for each file
  - MyRocks
    - Implements ACID + SQL on RocksDB
    - Uses lock table of RocksDB

Recap of lecture 7 (continued)

- **Partitioning**
  - Partitioning trade off
    - More nodes involved in each query: execute queries faster
    - On the hand, requires more coordination, if one node fails can slow everything down
  - Round robin partitioning
    - Balances load
    - All nodes involved
  - Hash partitioning
    - Usually balances load
    - All nodes involved
    - Can be used in distributed settings (without a central coordinator)
  - Range partitioning
    - Can be skewed
    - Few nodes involved
- **Skew**
  - Data skew
    - Data is not distributed evenly (e.g., using range partitioning on attribute, and certain values are more likely than others)
  - Execution skew
    - Certain data is accessed more frequently by a workload (e.g., certain rows are more “popular”/”hot”)

Today: Distributed systems

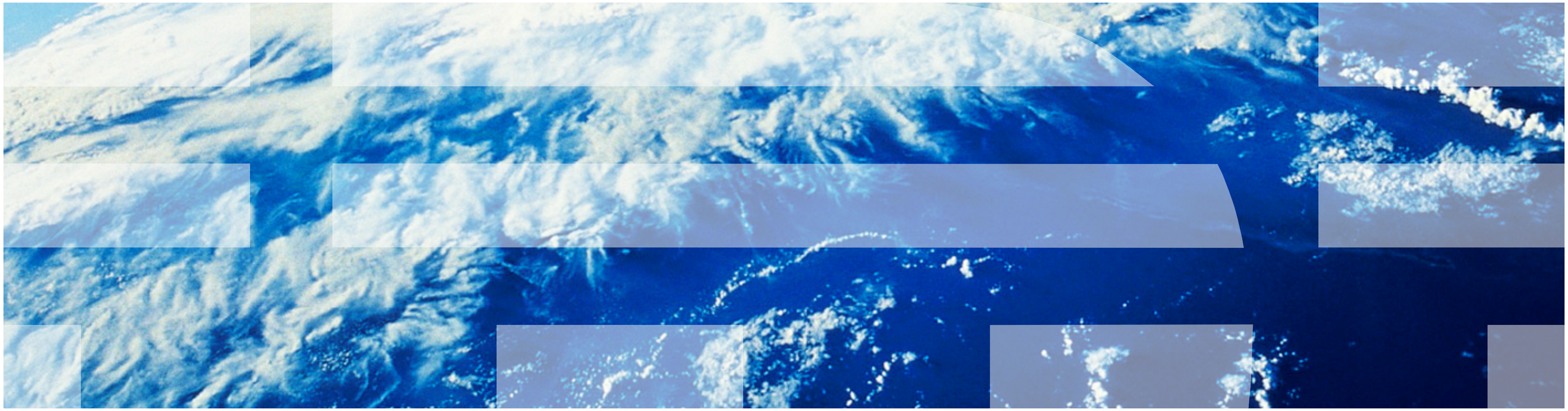
- Centralized master model
- Distributed file systems
  - Highlight: Hadoop File System (HDFS)
- Distributed transactions and databases
  - Two phase commit
  - Improvements of two phase commit

- If we have time:
- MapReduce
  - Programming model
  - Dealing with failures
- Spark intro

Logistics

- Midterm
  - You still have time to take it until tomorrow at noon
- Programming assignment #2
  - Should be available
  - Part 2 (streaming) is now optional
    - If you submit it, we will grade it, if not we won't
    - Recommended if you can –- nice introduction to streaming

Master node



Routing of Queries

- Partition table typically stored at a **master** node
- Two alternative designs:
  1. Queries are sent first to **master**, which forwards them to appropriate node
     - Disadvantage: need to communicate with master for each query
  2. Master tells **client** (the node asking the query) which nodes stores which key range → clients directly communicate with data nodes
     - Advantage: do not need to talk to master for each query
     - Disadvantages: what happens if a node dies and client has old information? Scalability
  - Examples of systems that use a master node: Hadoop File System, Google File System, BigTable (precursor to BigQuery), HBase (open source version of BigTable)
- **Consistent hashing** is an alternative fully-distributed scheme, without a master, uses a form of hash partitioning
  - Without any master nodes, works in a completely peer-to-peer fashion
  - Advantage: no single point of failure (master), scalability
  - Disadvantage: typically requires more communication/coordination among nodes
  - Examples of systems that use consistent hashing: Cassandra, DynamoDB

Replication

- Goal: **availability** despite failures
- Data replicated at 2, often 3 nodes
  - Why 3?
- Unit of replication typically a partition (tablet)
- Requests for data at failed node automatically routed to a replica
- Partition table with each tablet replicated at two nodes

Value	Tablet ID	Node ID
2012-01-01	Tablet0	Node0,Node1
2013-01-01	Tablet1	Node0,Node2
2014-01-01	Tablet2	Node2,Node0
2015-01-01	Tablet3	Node2,Node1
2016-01-01	Tablet4	Node0,Node1
2017-01-01	Tablet5	Node1,Node0
2018-01-01	Tablet6	Node1,Node2
MaxDate	Tablet7	Node1,Node2

Basics: Data Replication

- Location of replicas
  - **Replication within a data center**
    - Handles machine failures
    - Reduces latency if copy available locally on a machine
    - Replication within/across racks
  - **Replication across data centers**
    - Handles data center failures (power, fire, earthquake, ..), and network partitioning of an entire data center
    - Provides lower latency for end users if copy is available on nearby data center

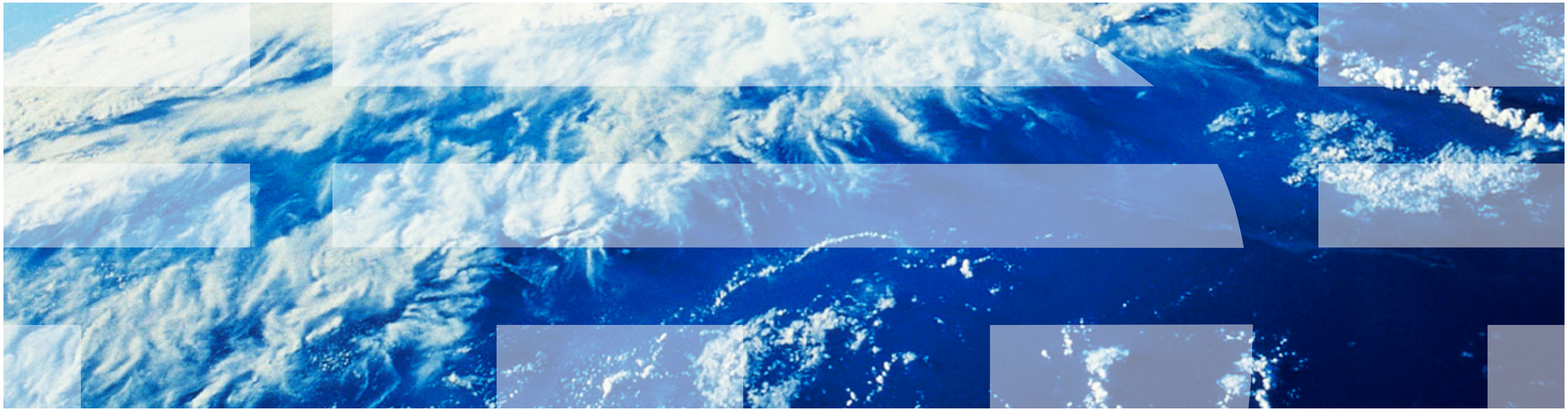
Updates and Consistency of Replicas

- Replicas must be kept consistent on update
  - Despite failures resulting in different replicas having different values (temporarily), reads must get the latest value.
  - Special concurrency control and atomic commit mechanisms to ensure consistency
- **Master replica** (**primary copy**) scheme
  - All updates are sent to master, and then replicated to other nodes
  - Reads are performed at master
  - But what if master fails? Who takes over? How do other nodes know who is the new master?

Protocols to Update Replicas

- *Two-phase commit*
  - Coming up!
  - Assumes all replicas are available
- *Consensus protocols*
  - Protocol followed by a set of replicas to agree on what updates to perform in what order
  - **Can work even without a designated master**

Distributed File Systems

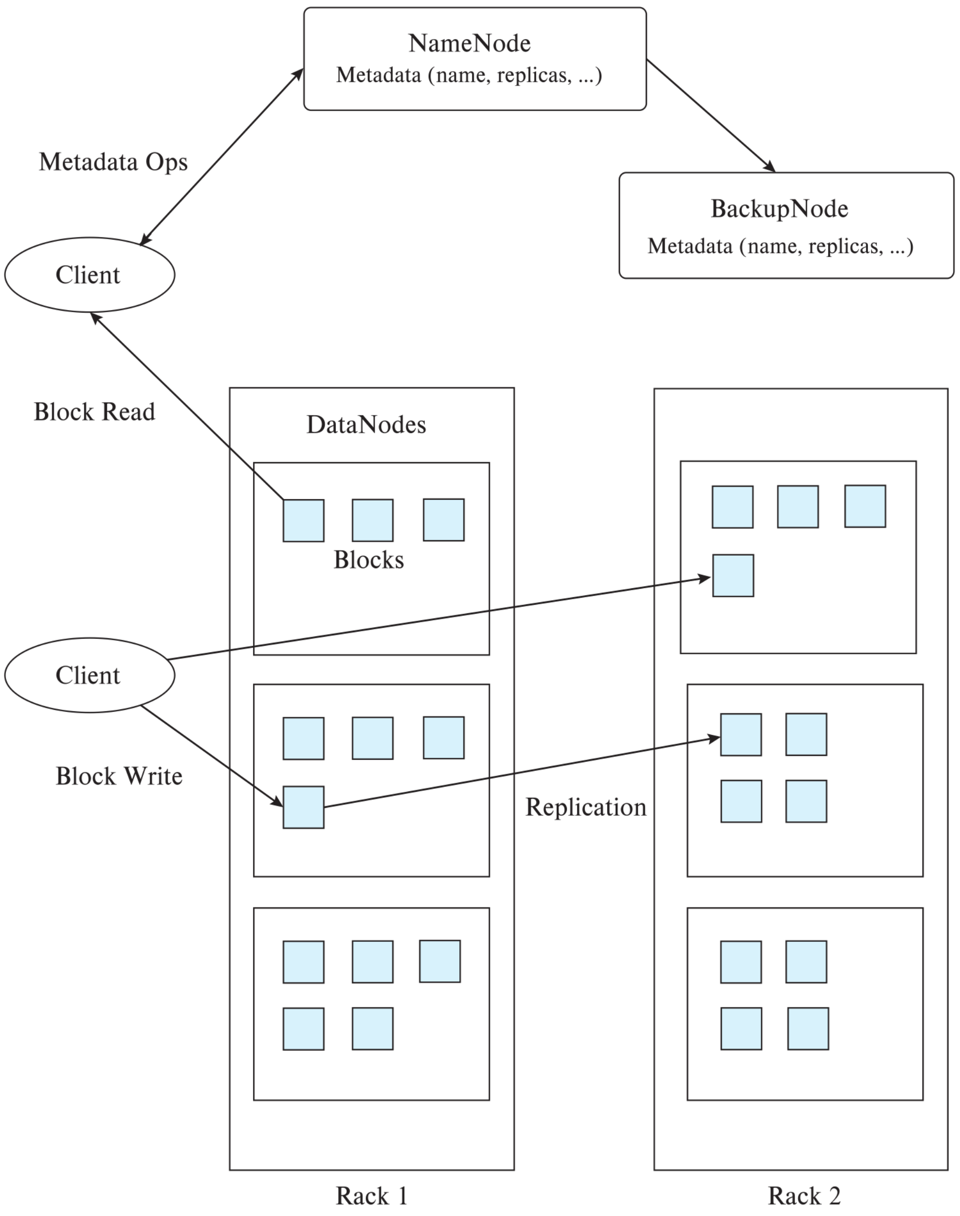


Distributed File Systems

- **Hadoop File System (HDFS)**
- Google File System (GFS)
- And older ones like CODA
- And more recent ones such as Google Colossus
- Basic architecture:
  - Master: responsible for metadata
  - Chunk servers: responsible for reading and writing large chunks of data
  - Chunks replicated on 3 machines, master responsible for managing replicas
  - Replication is in GFS/HDFS is within a single data center

Hadoop File System (HDFS)

- Client: sends filename to NameNode
- NameNode (the master)
  - Maps a filename to list of Block IDs
  - Maps each Block ID to DataNodes containing a replica of the block
  - Returns list of BlockIDs along with locations of their replicas
- DataNode:
  - Maps a Block ID to a physical location on disk
  - Sends data back to client



Hadoop Distributed File System

Hadoop Distributed File System (HDFS)

- Modeled after Google File System (GFS)
- Single Namespace (e.g., single directory structure) for entire cluster
- Data model
  - Write-once-read-many access model
  - Client can only append to existing files
- Files are broken up into blocks
  - Typically 64 MB block size
  - Each block replicated on multiple (e.g., 3) DataNodes
- Client
  - Finds location of blocks from NameNode
  - Accesses data directly from DataNode
    - NameNode is not on the critical path of reads and writes

Limitations of HDFS

- Central master becomes bottleneck
  - Keep directory information in memory to avoid expensive storage reads/writes
  - Memory size limits number of files
  - What happens if it fails?
- File system directory overheads per file
  - Not appropriate for storing very large number of objects
- File systems do not provide consistency guarantees
  - File systems cache blocks locally
  - Ideal for write-once and append only data
  - Can be used as underlying storage for a data storage system
    - E.g., **BigQuery/BigTable** uses GFS/Colossus underneath, **Hbase/Spark** uses HDFS underneath

Distributed File Systems vs. Databases

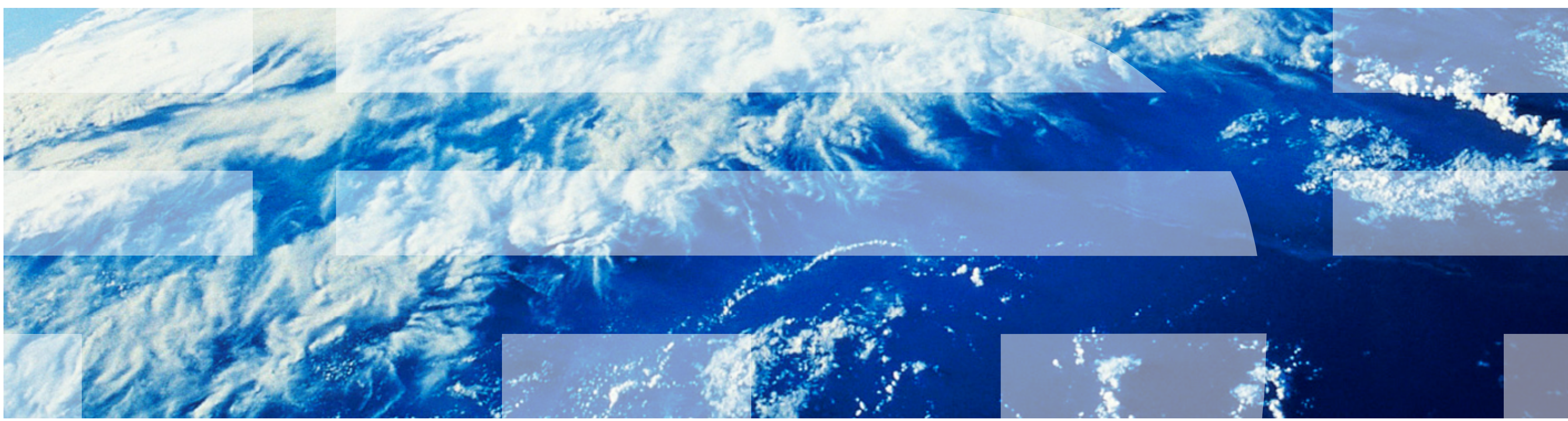
Distributed data storage implementations:

- May have limited support for relational model (no schema, or flexible schema)
- But usually do provide flexible schema and other features
  - Structured objects e.g. using JSON
  - Multiple versions of data items
- Often provide no support or limited support for transactions
  - But some do!
- Provide only lowest layer (similar to the file system layer)
- Often have KV store on top of them, and relational DB on top of the KV store

Geographically Distributed Storage

- Many storage systems today support geographical distribution of storage
  - Motivations: Fault tolerance, latency (close to user), governmental regulations
- Latency of replication across geographically distributed data centers much higher than within data center
  - Some key-value stores support **synchronous replication**
    - Must wait for replicas to be updated before committing an update
  - Others support **asynchronous replication**
    - update is committed in one data center, but sent subsequently (in a fault-tolerant way) to remote data centers
    - Must deal with small risk of data loss if data center fails.

Distributed Databases and Transactions



Approach 1: Sharding (AKA “shared-nothing” architecture)

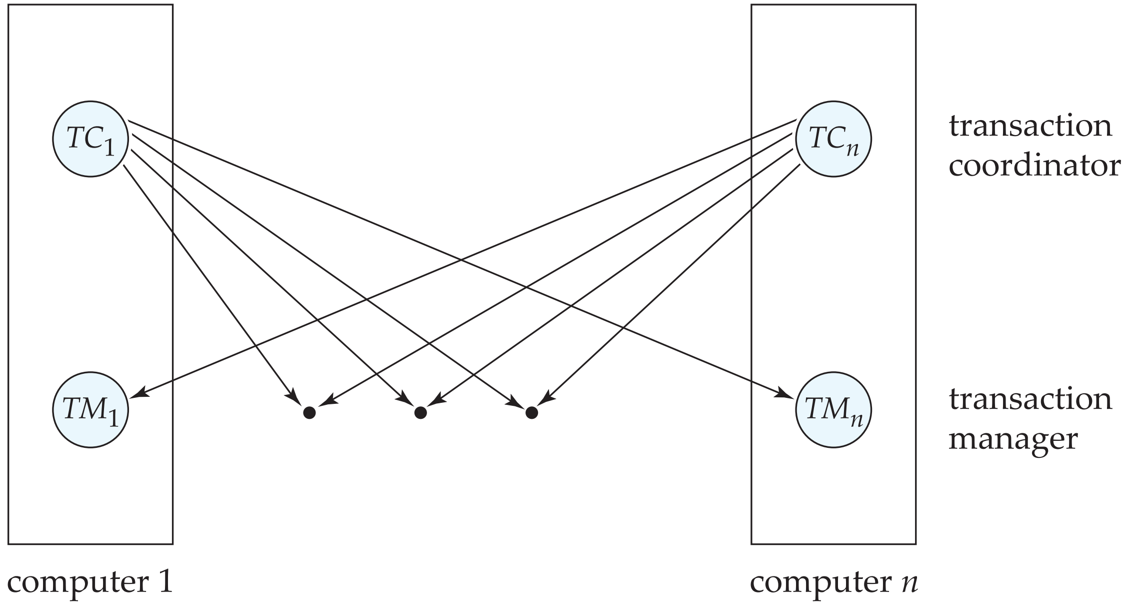
- Divide data amongst many cheap databases (MySQL/MyRocks/PostgreSQL)
- Manage parallel access in the application
  - Partition tables map keys to nodes
  - Application decides where to route storage or lookup requests
- Scales well for both reads and writes: used widely in data center settings
- Limitations
  - Not transparent
    - application needs to be partition-aware
    - AND application needs to deal with replication
  - (Not a true parallel database, since parallel queries and transactions spanning nodes are not supported)

Approach 2: Distributed Transactions

- **Local transactions**
  - Access/update data at only one database
- **Global transactions**
  - Access/update data at more than one database
- Key issue: how to ensure ACID properties for transactions in a system with global transactions spanning multiple database

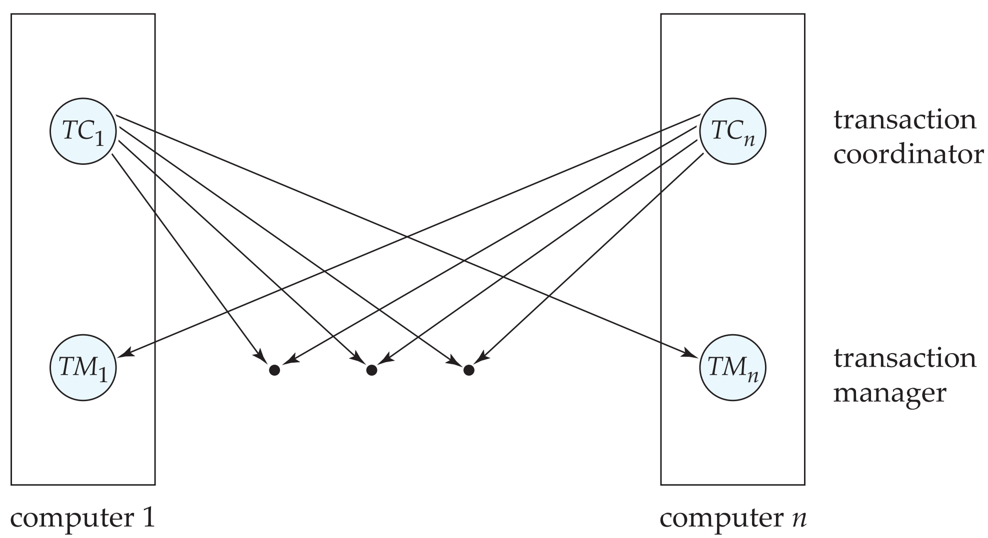
Distributed Transactions

- Transaction may access data at several sites.
  - Each site has a local **transaction manager**
  - Each site has a **transaction coordinator**
    - Global transactions submitted to any transaction coordinator



Distributed Transactions

- Each transaction coordinator is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site
    - Transaction must be committed at all sites or aborted at all sites.
- Each local transaction manager responsible for:
  - Maintaining a log for recovery purposes for the node it belongs to
  - Coordinating the execution and commit/abort of the transactions executing at that site.



System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site
  - Loss of messages
    - Handled by networking protocols such as TCP-IP
  - **Network partition**
    - A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them

- Network partitioning and site failures are generally indistinguishable.

Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - A transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
    - Cannot have transaction committed at one site and aborted at another
    - Why?
- **The *two-phase commit* (2PC) protocol is widely used**
- *Consensus protocols* solve a more general problem, but can be used for atomic commit
- We assume **fail-stop** model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites (they do not become “malicious”)
  - This is the assumption in many distributed systems, but not all
  - For example, Blockchains assume that each node can be malicious (**Byzantine failure model**)
  - Generally a reasonable assumption when all the nodes reside under the same organization, with no conflicting interests

Two Phase Commit Protocol (2PC)

- Execution of the protocol is initiated by any coordinator.
- The protocol involves all the local sites at which the transaction executed
- Protocol has two phases
- Let *T* be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

Phase 1: Obtaining a Decision

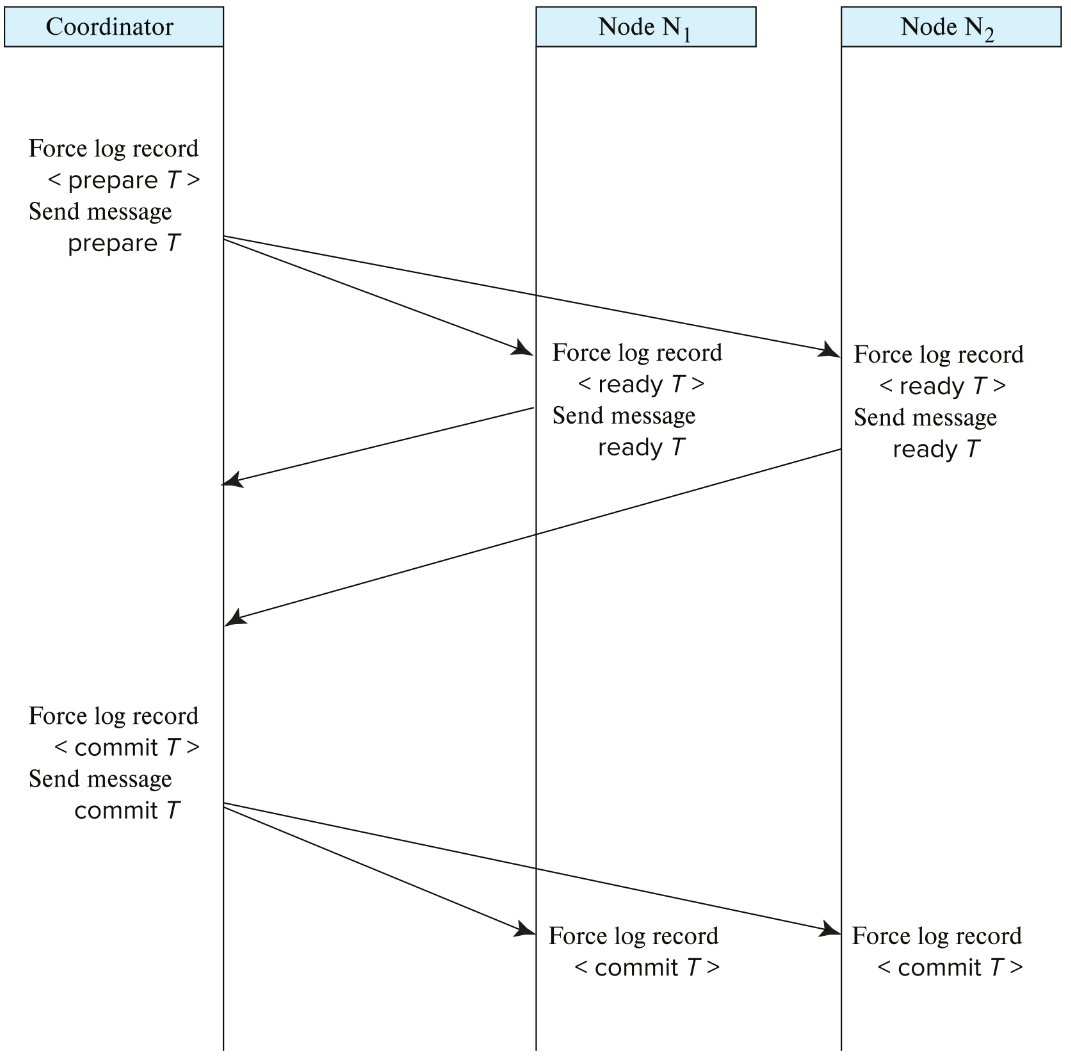
- Coordinator asks all participants to *prepare* to commit transaction T_i.
  - C_i adds the records <**prepare** *T*> to the log and forces log to stable storage
  - sends **prepare** *T* messages to all sites at which *T* executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <**no** *T*> to the log and send **abort** *T* message to C_i
  - if the transaction can be committed, then:
    - add the record <**ready** *T*> to the log
    - force *all records* for *T* to stable storage (i.e., execute the transaction locally)
    - send **ready** *T* message to C_i

    Transaction is now in ready state at the site

Phase 2: Recording the Decision

- *T* can be committed if C_i received a **ready** *T* message from all the participating sites: otherwise *T* must be aborted.
- Coordinator adds a decision record, <**commit** *T*> or <a**bort** *T*>, to the log and forces record onto stable storage. Once the record is in stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.
  - If the transaction was committed, they can commit locally
  - If it was aborted, they need to undo the transaction based on their local logs

Two-Phase Commit Protocol



Handling of Failures - Site Failure

When site S_k recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain <**commit** *T*> record: site executes **redo** (*T*)
  - Uses the WAL
- Log contains <**abort** *T*> record: site executes **undo** (*T*)
  - Uses the WAL
- Log contains <**ready** *T*> record: site must consult C_i to determine the fate of *T*.
  - If *T* committed, **redo** (*T*)
  - If *T* aborted, **undo** (*T*)
- The log contains no control records concerning *T* implies that S_k failed before responding to the **prepare** *T* message from C_i
  - since the failure of S_k precludes the sending of such a response C_i must abort *T*
  - S_k must execute **undo** (*T*)

Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*'s fate:
  1. If an active site contains a <**commit** *T*> record in its log, then *T* must be committed.
  2. If an active site contains an <**abort** *T*> record in its log, then *T* must be aborted.
  3. If some active participating site does not contain a <**ready** *T*> record in its log, then the failed coordinator C_i cannot have decided to commit *T*. Can therefore abort *T*.
  4. If none of the above cases holds, then all active sites must have a <**ready** *T*> record in their logs, but no additional control records (such as <**abort** *T*> of <**commit** *T*>). In this case active sites must wait for C_i to recover, to find decision.
- **Blocking problem**: active sites may have to wait for failed coordinator to recover.

Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator.
- The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
    - Again, no harm results

Recovery and Concurrency Control

- **In-doubt transactions** have a <**ready** *T*>, but neither a <**commit** *T*>, nor an <**abort** *T*> log record.
- The recovering site must determine the commit-abort status of such transactions by contacting other sites; this can slow and potentially block recovery.
- Recovery algorithms can note lock information in the log.
  - Instead of <**ready** *T*>, write out <**ready** *T*, *L*>
    - *L* = list of locks held by *T* when the log is written (read locks can be omitted).
  - For every in-doubt transaction *T*, all the locks noted in the <**ready** *T*, *L*> log record are reacquired.
- After lock reacquisition, transaction processing can resume; the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions.

Avoiding Blocking During Consensus

- Blocking problem of 2PC is a serious concern
  - **Any** participant that fails or is delayed will block all other nodes!
- Idea: involve multiple nodes in decision process, so failure of a few nodes does not cause blocking as long as majority don't fail
- More general form: **distributed consensus problem**
  - A set of *n* nodes need to agree on a decision
  - Inputs to make the decision are provided to all the nodes, and then each node votes on the decision
  - The decision should be made in such a way that all nodes will "learn" the same value for the even if some nodes fail during the execution of the protocol, or there are network partitions.
  - Further, the distributed consensus protocol should not block, as long as a **majority** of the nodes participating remain alive and can communicate with each other
- Several consensus protocols, Paxos and Raft are popular

Using Consensus to Avoid Blocking

- After getting response from 2PC participants, coordinator can initiate distributed consensus protocol by sending its decision to a set of participants who then use consensus protocol to commit the decision
  - If coordinator fails before informing all consensus participants
    - Choose a new coordinator, which follows 2PC protocol for failed coordinator
    - If a commit/abort decision was made as long as a majority of consensus participants are accessible, decision can be found without blocking
  - If consensus process fails (e.g., split vote), restart the consensus
    - Split vote can happen if a coordinator send decision to some participants and then fails, and new coordinator send a different decision
- The **three phase commit** protocol is an extension of 3PC which avoids blocking under certain assumptions
  - Ideas are similar to distributed consensus.
- Consensus is also used to ensure consistency of replicas of a data item
  - For example, can be used to protect against failure of master nodes (e.g., the NameNode in HDFS)